Hypercluster Parallel Processing Library User's Manual

Version 1.0
March 1990

Prepared For:
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Developed By:
Sverdrup Technology, Inc.
Lewis Research Center Group
2001 Aerospace Parkway
Brook Park, Ohio 44142

CONTENTS

CONTENTS

INTRODUCTION

The Hypercluster at NASA-Lewis

The Hypercluster is a modification of the popular hypercube architecture. Each node of the Hypercluster consists of a cluster of processors, rather than a single processor, as illustrated in Figure 1 (see page 65). Each processor is identified by a node ID and a processor ID. Several computational processors per node are available to run applications, and vector processors may be accessible to the computational processors. Several processors are also used only for communication.

A message passing kernel (MPK) resides on each processor, and controls communication among processors. FORTRAN runs on top of the MPK, and the FORTRAN-callable subroutines in the parallel processing library invoke MPK operations to manipulate information among processors.

Each processor cluster can communicate within its own cluster using shared memory. A shared memory area must be established (using a parallel processing library subroutine) in order to be accessed from a FORTRAN program. Synchronization is necessary among processors accessing shared memory, so several synchronization alternatives are offered in the library.

Processors located in different clusters must communicate using message passing. Messages are routed throughout the Hypercluster using the hypercube topology. Parallel processing library subroutines allow each Hypercluster processor to access any other processor in the system. This architecture allows the programmer to explore both shared memory and distributed memory operations, as well as a combination of the two, within a single programming environment.

The Parallel Processing Library

A Parallel Processing Library has been developed to assist the FORTRAN programmer working with the Hypercluster. This library consists of FORTRAN-callable subroutines which enable the user to manipulate and transfer information throughout the Hypercluster. The subroutines in this library are currently the only method available to FORTRAN programmers to communicate program data between Hypercluster processors.

The Parallel Processing Library is divided into three areas. The first area involves operations to establish and manipulate shared memory. The second area involves operations to manipulate information in a distributed memory environment. The third area involves miscellaneous operations such as manipulating a timer, or identifying a processor. Each of these areas is described in detail in a separate section of this manual.

The last section of this manual includes a simple programming example which employs many of the subroutines from the Parallel Processing Library. This sample problem may assist FORTRAN programmers in converting their own codes for execution on the Hypercluster.

SHARED MEMORY SUBROUTINES

Establishing Areas of Shared Memory

A shared memory area may be established within a node of the Hypercluster. A processor can establish a connection to a shared memory area on its node, and can access the information stored there as if the information were located in its own local memory. This allows processors within a node to exchange information, enabling parallel processing within a node.

A shared memory area can only be established among computational processors. Variables to be shared are located in the first common block that is declared in each program segment which uses one of the shared variables. A parallel processing library subroutine, ASHRDL, establishes a connection to the shared memory area and specifies on which processor a shared memory area is to be located. The entire shared common block is located on that specified processor; it cannot be divided so that some variables are stored on one processor and some variables are stored on another processor.

In the example below, program CALCTMP is executed on processor 1. The variables TEMP, PR, and DENS are stored and accessed from the local memory of processor 3.

P1

```
      PROGRAM CALCTMP
      COMMON /SHRDV/ TEMP,PR,DENS
      COMMON /OTHER/ X,Y,Z,TIME
C
      CALL ASHRDL (3)
      .
      .
      .
      CALL DSHRDL (3)
C
      STOP
      END
```

Any link to a shared memory area which is established using ASHRDL must be eliminated using DSHRDL before exiting a program segment. This action is necessary to maintain FORTRAN's internal representation of common blocks.

Synchronization

All processors who have access to a shared memory area can potentially access that shared area simultaneously. Synchronization is necessary to control shared memory accesses, guaranteeing that information is updated appropriately.

The following synchronization routines are available in the Parallel Processing Library:

- SYNC, which is a barrier synchronization method
- LOCK and UNLOCK, which provide for protecting critical sections of code
- POSTEV, WAITEV, and CLREV, which provide a method of dictating the order of events which occur across multiple processors.

Subroutines

A detailed description of each shared memory subroutine included in the Parallel Processing Library follows.

SUBROUTINE: **ASHRDL**

FORMAT: CALL ASHRDL (pid)

pid Integer variable or constant specifying the processor on which a shared memory area is to be located. It can be any computational processor on the node.

DESCRIPTION:

ASHRDL establishes a link to a shared memory area on the processor specified by parameter **pid.** This link is established within a particular program segment, defined as a main program or a subroutine. Any program segment which accesses a shared variable must call ASHRDL to establish a link to the shared memory area.

A shared memory area is defined by the first declared common block in a program segment. All accesses to variables in that common block are made to the shared area on processor pid. Processor pid owns the shared data area, but once ASHRDL is called, the calling processor has access to the shared data as if the data were its own local data. If the calling processor is pid, all accesses to variables in that common block are made to its own memory.

In the example on page 07, program CALCTMP is executed on processor 1, and program CALCPR is executed on processor 3. The shared memory area defined by common block /SHRDV/ is located on processor 3. All references to variables TMP, PR, and DNS in the main program of CALCTMP, or T, P, and D in SUBR1 are made to processor 3's copy of these variables.

Any link to a shared memory area which is established in a program segment using ASHRDL must be eliminated in that program segment using subroutine DSHRDL, as demonstrated in the example.

SUBROUTINE: **DSHRDL**

FORMAT: CALL DSHRDL (pid)

pid Integer variable or constant specifying the processor on which a shared memory area is located. It must be the same computational processor on which the shared region was previously established using subroutine ASHRDL.

DESCRIPTION:

DSHRDL eliminates a link to a shared memory area on the processor specified by parameter **pid**. This link was previously established by the ASHRDL subroutine. Any program segment which established a link to a shared data area must call DSHRDL to eliminate that link.

An example follows.

EXAMPLE for ASHRDL and DSHRDL

P1

```
      PROGRAM CALCTMP
      COMMON /SHRDV/ TMP,PR,DNS
      COMMON /OTHER/ X, Y, Z, T

C Establish a link to the shared area
C SHRDV on processor 3

      CALL ASHRDL (3)
          .
          .

      TMP= A+B
      X= PR*C
          .
          .
      CALL SUBR1
          .
          .
C Eliminate link to SHRDV
      CALL DSHRDL (3)
      STOP
      END

      SUBROUTINE SUBR1
      COMMON /SHRDV/ T, P, D

C This is a new program segment,
C so establish a link to SHRDV
C on processor 3
      CALL ASHRDL (3)
          .
          .

      X1= D*(A1+A2)

          .
C Eliminate link to SHRDV
      CALL DSHRDL (3)
      RETURN
      END
```

P3

```
      PROGRAM CALCPR
      COMMON /SHRDV/ T1, P1, D1

C Establish a link to the shared area
C SHRDV on processor 3

      CALL ASHRDL (3)
          .
          .

      D1= C+D*E
      T1= D+F

          .
          .
C Eliminate link to SHRDV
      CALL DSHRDL (3)
      STOP
      END
```

SUBROUTINE: **SYNC**

FORMAT: CALL SYNC (num, idlist)

num Integer variable or constant specifying the number of processors to synchronize.

idlist Integer array specifying the processor ids to be synchronized (includes the calling processor). All processors in **idlist** must be computational processors within a single node.

DESCRIPTION:

SYNC provides a synchronization facility to maintain shared memory activities within a node. It is a barrier-type synchronization operation, where **num** processors specified by **idlist** must come to a common synchronization point before any are allowed to continue execution. A timeout mechanism is built into this synchronization to avoid deadlock. If a timeout situation should occur, an advisory message is sent to the operating system, and program execution terminates.

This method is used to synchronize activities among processors. For example, only one processor should initialize a shared variable. Other processors on the node should not access the variable before it is initialized. SYNC is used to detain other processors until the initialization is complete, as illustrated in the example which follows.

EXAMPLE for SYNC

P1

```
      PROGRAM CALCTMP
      COMMON /SHRDV/ TMP,PR,DNS
      COMMON /OTHER/ X,Y,Z,TIME
      DIMENSION IVAR(2)

C     Establish a link to the shared
C     area SHRDV on processor 3
      CALL ASHRDL (3)

C     Synchronize with proc 3
C     (waiting for 3 to
C     initialize shared vars)
      IVAR(1)= 1
      IVAR(2)= 3
      CALL SYNC (2,IVAR)
          .
          .
          .
C     Eliminate link to SHRDV
      CALL DSHRDL (3)
      STOP
      END
```

P3

```
      PROGRAM CALCPR
      COMMON /SHRDV/ T1,P1,D1
      DIMENSION IVAR(2)

C     Establish a link to the shared area
C     SHRDV on processor 3
      CALL ASHRDL (3)

C     Initialize shared variables
      T1= 98.6
      P1= 1.0
      D1= 4.6

C     Synchronize with proc 1
      IVAR(1)= 1
      IVAR(2)= 3
      CALL SYNC (2,IVAR)
          .
          .
C     Eliminate link to SHRDV
      CALL DSHRDL (3)
      STOP
      END
```

SUBROUTINE: **LOCK**

FORMAT: CALL LOCK (isem)

isem Integer variable used as a semaphore.

DESCRIPTION:

LOCK provides synchronization to protect a critical region in shared memory. It is used with the UNLOCK subroutine. Both subroutines operate on a special type of variable called a semaphore.

A semaphore, **isem**, is shared among the processors which operate on it. Subroutine LOCK guarantees that only one processor on a node has access to isem at one time. Once a processor gains access to isem, all other processors wanting access must idle until isem is UNLOCKed. The LOCKing processor essentially blocks others until it is finished with what it is doing. This allows one processor to gain access to a critical region and manipulate shared variables without interference from other processors. When finished, isem is UNLOCKed, allowing a waiting processor to access the critical region.

Any processors waiting for access to isem sit in an idle loop, waiting for isem to be UNLOCKed. A timeout mechanism is built into this idle loop to avoid deadlock. If a timeout situation should occur, an advisory message is sent to the operating system, and program execution terminates.

For an example, see page 12.

SUBROUTINE: **UNLOCK**

FORMAT: CALL UNLOCK (isem)

isem Integer variable used as a semaphore.

DESCRIPTION:

UNLOCK signals that a processor has exited a critical region. It is used in conjunction with subroutine LOCK. Both subroutines operate on a special type of variable called a semaphore. A semaphore, **isem**, is shared among the processors which operate on it. UNLOCK clears semaphore isem, enabling another processor to gain access using subroutine LOCK. The critical region which was protected by isem is now available to the next processor attempting to make access.

An example follows.

EXAMPLE for LOCK and UNLOCK

P1

```
      PROGRAM MAIN1
      COMMON /COM/ ISEM,N,NH
      DIMENSION IVAR(2)

C     Establish a link to the shared
C     area COM on processor 3
      CALL ASHRDL (3)

C     Synchronize with proc 3
C     (waiting for 3 to
C     initialize shared vars)
      IVAR(1)= 1
      IVAR(2)= 3
      CALL SYNC (2,IVAR)

C     Perform first half of
C     iteration, steps 1..NH
      DO 10 J= 1, NH
         .
         .
C     Attempt to access critical section
      CALL LOCK (ISEM)
         .
      (execute code in
      critical section)
         .
         .
C     Exit critical section
      CALL UNLOCK (ISEM)
         .
         .
10    CONTINUE

C     Eliminate link to COM
      CALL DSHRDL (3)
      STOP
      END
```

P3

```
      PROGRAM MAIN2
      COMMON /COM/ ISEM,N,NH
      DIMENSION IVAR(2)

C     Establish a link to the shared area
C     COM on processor 3
      CALL ASHRDL (3)

C     Initialize shared variables
      ISEM= 0
      N= 100
      NH= N/2

C     Synchronize with proc 1
      IVAR(1)= 1
      IVAR(2)= 3
      CALL SYNC (2,IVAR)

C     Perform second half of
C     iteration, steps NH+1..N
      DO 10 J= NH+1, N
         .
C     Attempt to access critical section
      CALL LOCK (ISEM)
         .
         .
      (execute code in
      critical section)
         .
         .
C     Exit critical section
      CALL UNLOCK (ISEM)
         .
         .
10    CONTINUE

C     Eliminate link to COM
      CALL DSHRDL (3)
      STOP
      END
```

SUBROUTINE: **POSTEV**

FORMAT: CALL POSTEV (ievt)

ievt	Integer variable used as an event flag. The value of this flag can be "posted" or "cleared."

DESCRIPTION:

POSTEV provides synchronization which allows a programmer to dictate the order of activities (or "events") which occur across multiple processors. This subroutine is used in conjunction with WAITEV and CLREV. Each of these subroutines operates on a special type of variable called an event flag. An event flag, **ievt**, is shared among the processors which operate on it.

An event is an occurrence of some specific event within a program's execution. For example, one processor may calculate a value which is required by other processors. That processor posts the event "calculation complete." Other processors waiting on this event (WAITEV) can then proceed. Note that it is not logical to re-post an event which has already been posted.

For an example, see page 16.

SUBROUTINE: **WAITEV**

FORMAT: CALL WAITEV (ievt)

ievt Integer variable used as an event flag. The value of this flag can be "posted" or "cleared."

DESCRIPTION:

WAITEV provides synchronization which allows a programmer to dictate the order of activities (or "events") which occur across multiple processors. This subroutine is used in conjunction with POSTEV and CLREV. Each of these subroutines operates on a special type of variable called an event flag. An event flag, **ievt**, is shared among the processors which operate on it.

A processor waits on an event which is posted by another processor. (Note that it is not logical for a processor to wait on an event which it posts itself.) If an event has been posted, control is returned to the calling program. When an event has not been posted, the processor idles until that event is posted. A timeout mechanism is built into this idle loop to avoid deadlock. If a timeout should occur, an advisory message is sent to the operating system, and program execution terminates.

For an example, see page 16.

SUBROUTINE: **CLREV**

FORMAT: CALL CLREV (ievt)

ievt	Integer variable used as an event flag. The value of this flag can be "posted" or "cleared."

DESCRIPTION:

CLREV provides synchronization which allows a programmer to dictate the order of activities (or "events") which occur across multiple processors. This subroutine is used in conjunction with POSTEV and WAITEV. Each of these subroutines operates on a special type of variable called an event flag. An event flag, **ievt**, is shared among the processors which operate on it.

Once an event has been posted, and all waiting processors have proceeded, that event should be cleared. This is especially significant when the manipulation of event flags occurs within a loop, as demonstrated in the example that follows. A previously posted event must be cleared before the next iteration re-posts to that same event.

EXAMPLE for POSTEV, WAITEV, and CLREV

P1

```
      PROGRAM MAIN1
      COMMON /COM/ IEV1,IEV2,X
      DIMENSION IVAR(2)

C     Establish a link to the shared area
C     COM on processor 1
      CALL ASHRDL (1)

C     Initialize shared variables
      IEV1= 0
      IEV2= 0
      X= 100.0

C     Synchronize with proc 2
      IVAR(1)= 1
      IVAR(2)= 2
      CALL SYNC (2,IVAR)

      DO 10 J= 1,100,2
         .
         .
      (perform EVENT #1)
C     Signal completion of event #1
      CALL POSTEV (IEV1)

C     Wait for event #2 to
C     complete
      CALL WAITEV (IEV2)

C     Clear event #2 for next
C     iteration
      CALL CLREV (IEV2)

10    CONTINUE

C     Eliminate link to COM
      CALL DSHRDL (1)

      STOP
      END
```

P2

```
      PROGRAM MAIN2
      COMMON /COM/ IEV1,IEV2,X
      DIMENSION IVAR(2)

C     Establish a link to the shared area
C     COM on processor 1
      CALL ASHRDL (1)

C     Synchronize with proc 1
C     (waiting for 1 to
C     initialize shared vars)
      IVAR(1)= 1
      IVAR(2)= 2
      CALL SYNC (2,IVAR)

      DO J= 2,100,2

C     Wait for event #1 to complete
      CALL WAITEV (IEV1)

C     Clear event #1 for next iteration
      CALL CLREV (IEV1)
         .
         .

      (perform EVENT #2)
C     Signal completion of event #2
      CALL POSTEV (IEV2)

10    CONTINUE

C     Eliminate link to COM
      CALL DSHRDL (1)

      STOP
      END
```

DISTRIBUTED MEMORY SUBROUTINES

Methods of Transferring Distributed Information

Because the Hypercluster is also a distributed memory architecture, a method of communicating information between distributed nodes is provided. The Parallel Processing Library provides two methods of distributed information transfer.

The first type is an indirect, synchronous transfer, which involves both the sender and the receiver in the transfer of information. The exact address of the source or destination of the information is unknown. Subroutines to perform this type of transfer include:

- OPENCH
- SEND
- SENDB
- RECV
- RECVW
- BRDCST

The second type of distributed information transfer is a direct, asynchronous transfer, which involves only the initiator in the transfer of information. In this situation, the exact source and destination locations are known. It is the responsibility of the FORTRAN programmer to determine whether the transfer is complete before accessing the information involved. Subroutines to perform this type of transfer include:

- READM
- WRITEM

These distributed information transfer methods are most often used between processors on different nodes, although they can be used between processors within a node. Each type of transfer is described individually below, along with a description of the subroutines which perform that type of transfer.

Parameters and Restrictions

Most distributed memory subroutines require the number of bytes being transferred as a parameter. Table 1 lists each FORTRAN data type, along with its corresponding number of bytes (see page 63).

One restriction on distributed memory transfers involves variables in shared memory. Certain distributed memory subroutines (SEND, READM, WRITEM) will not perform correctly when transferring variables in shared memory. The safest method to avoid transfer problems in this situation is to allow only the processor on which a shared area is located to transfer a variable in shared memory.

Indirect Information Transfers

Indirect information transfers provide a synchronous method of transferring information among processors in the Hypercluster. This method requires both the sender and the receiver to be involved in the transfer of information.

A logical link is established between the processors intending to transfer information. This logical link is referred to as a channel. Once a channel is established, information may be sent or broadcast to another processor who has access to that channel. That processor may then receive the information when needed.

Two types of sends (SEND, SENDB) and two types of receives (RECV, RECVW) are included in the Parallel Processing Library. The difference between the send subroutines involves where the information is located during the transfer. The difference between the receive subroutines involves the subroutine response when information is unavailable.

Direct Information Transfers

Direct information transfers provide an asynchronous method for transferring information among processors in the Hypercluster. This method takes advantage of the MPK's built-in ability to read and write information between any two processors in the configuration.

A direct information transfer requires the physical location of the information source and destination, which is generally unavailable to FORTRAN programmers. To overcome this obstacle, FORTRAN programmers can use the structure of FORTRAN common blocks to their advantage, to "fool" the system into thinking they know the proper physical locations of the information being transferred.

The example below illustrates this technique. Common block A on node 1, processor 2 (N1P2) and common block B on N0P3 are located at the same physical address in their respective memories. Common block C on N1P2 and common block D on N0P3 are located at the same physical address in their respective memories. This is because of the way FORTRAN allocates common block storage. Thus the address of variable X on N1P2 is the same as the address of variable C on N0P3. The address of array U on N1P2 is the same as the address of array F on N0P3. In the example below, the WRITEM subroutine writes the value of X on N1P2 to variable H on N0P3. READM reads array F on N0P3, and stores it in array U on N1P2.

N1P2	N0P3
PROGRAM MAIN1	PROGRAM MAIN2
COMMON /A/ X,Y,Z(100)	COMMON /B/ C,H,E(100)
COMMON /C/ T,U(5),V	COMMON /D/ A,F(5),G
CALL WRITEM (X,0,3,Y,4,IFLG)	.
CALL READM (0,3,U,U,20)	.
STOP	STOP
END	END

It must be remembered that these routines are asynchronous, so more responsibility is required from the programmer to guarantee that information exists before it is accessed, or that information has been transferred before it is altered.

Subroutines

A detailed description of each distributed memory subroutine included in the Parallel Processing Library follows.

SUBROUTINE: **OPENCH**

FORMAT: CALL OPENCH (channel, node id, proc id)

channel Integer variable or constant specifying the identification number of a logical link to another processor. This other processor is defined by parameters **node id** and **proc id.**

node id Integer variable or constant specifying node id.

proc id Integer variable or constant specifying processor id.

DESCRIPTION:

OPENCH establishes a logical link, or **channel**, between the calling processor and the processor specified by parameters **node id** and **proc id.** A channel is required to send, receive, or broadcast a message through the Hypercluster. The closing of a channel is handled by the MPK.

There are 32 available channels, numbered 0 through 31. A channel used by the sender or broadcaster of a message must have the same id number as the channel used by the receiver(s) of the message. To avoid confusion in assigning channel numbers, a convention has been adopted where channel numbers 0,1,2,... will be assigned to messages being sent and received. Channel numbers 31,30,29,... will be assigned to messages being broadcast and received.

All processors involved in a specific broadcast must use the same channel id number. This channel number cannot be used by processors which are not involved in the broadcast. In particular, a channel which is set up for the purpose of a broadcast cannot be used for an individual send/receive. A "dummy" channel is established by the broadcasting processor. Non-valid entries of -1 are supplied as OPENCH parameters node id and proc id. These entries indicate that this processor is broadcasting a message to multiple processors on the same channel. Processors receiving a broadcast establish a valid channel, specifying the broadcaster's id as parameters node id and proc id.

An example follows.

EXAMPLE for OPENCH

```
      PROGRAM MAIN1

      CH=1
      NDID= 3
      PRID= 2

C     Open channel 1 to node 3, processor 2
      CALL OPENCH (CH,NDID,PRID)
          .
          .
          .

      STOP
      END
```

SUBROUTINE: **SEND**

FORMAT: CALL SEND (channel, msg, num, flag addr)

channel Integer variable or constant specifying the identification number of a logical link to another processor.

msg Variable to be transferred.

num Integer variable specifying the number of bytes to transfer.

flag addr Integer variable which is returned to the user. It contains the address of a word which is cleared once the transfer is initiated.

DESCRIPTION:

SEND synchronously transmits information from one processor to another. This subroutine must be coupled with a receive subroutine (RECV or RECVW) executed on the receiving processor. The **channel** declared by the sender of the information must be the same as the channel used by the receiver of the information. **Msg** can be any variable in the FORTRAN program. It can be an array name, or even a section of an array. **Num** indicates the number of bytes to transfer, and must be less than 64 kbytes. The number of bytes for various FORTRAN data types is listed in Table 1 (see page 63).

The sending processor can poll the word to which **flag addr** points to determine whether the transfer has been initiated. When this memory location is clear, the transfer has been initiated. SEND is distinguished from SENDB because the information being transferred using SEND is not physically included in the initial message. Thus the information transferred using SEND cannot be altered until the programmer is sure that the transfer has been initiated (i.e., the word pointed to by flag addr is clear). Note that the term word implies a two-byte memory location. The example below includes a test sequence which is recommended in order to poll this variable correctly. The function WORD is supplied by FORTRAN 77 (ABSOFT, 1986).

It must be noted that if SEND is used to transfer a variable in shared memory, then that shared data area must be located on the processor performing the SEND. If the shared variable is located elsewhere, SEND does not perform correctly. A SENDB can be used in this situation to avoid a transfer error.

An example follows.

EXAMPLE for SEND

N0P3

```
      PROGRAM MAIN1
      DIMENSION X(10)
      INTEGER*2 IVAL

        .
        .
      CALL OPENCH (2,3,2)
      CALL SEND (2,X,40,IFADDR)

C     Query to determine whether the
C     transfer has been initiated

 10   IVAL = WORD (IFADDR)
      IF (IVAL .EQ. 0)
      THEN WRITE (1,100)
100   FORMAT (" Message initiated")

      ELSE GOTO 10

C     Variable X can now be altered
C     if so desired
      X(1)= 100.23

      STOP
      END
```

N3P2

```
      PROGRAM MAIN2
      DIMENSION Y(10)

      CALL OPENCH (2,0,3)

      CALL RECVW (2,Y,40)

C     Variable Y is now available for use

      Z= 2*Y(3)-C
        .
        .


      STOP
      END
```

SUBROUTINE: **SENDB**

FORMAT: CALL SENDB (channel, msg, num)

channel Integer variable or constant specifying the identification number of a logical link to another processor.

msg Variable to be transferred.

num Integer variable or constant specifying the number of bytes to transfer.

DESCRIPTION:

SENDB synchronously transmits information from one processor to another. This subroutine must be coupled with a receive subroutine (RECV or RECVW) executed on the receiving processor. The **channel** declared by the sender of the information must be the same as the channel used by the receiver of the information. **Msg** can be any variable in the FORTRAN program. It can be an array name, or even a section of an array. **Num** indicates the number of bytes to transfer, and must be less than 64 kbytes. The number of bytes for various FORTRAN data types is listed in Table 1 (see page 63).

SENDB composes a message internally, containing a copy of the information to be transferred. Since the message contains a copy of the information, the programmer does not have to wait until the message is sent before altering this information. This distinguishes SENDB from SEND. Although costly in terms of having to make a copy of the information to be transferred, SENDB does not have to query to determine whether the transfer has been initiated in order to alter the information being sent.

An example follows.

EXAMPLE for SENDB

N0P3

```
      PROGRAM MAIN1
      DIMENSION X(10)

        .
        .
      CALL OPENCH (2,3,2)
      CALL SENDB (2,X,40)

C     Variable X can now be
C     altered if so desired
      X(1)= 100.23

      STOP
      END
```

N3P2

```
      PROGRAM MAIN2
      DIMENSION Y(10)

      CALL OPENCH (2,0,3)

      CALL RECVW (2,Y,40)

C     Variable Y can now be accessed

      Z= 2*Y(3)-C

        .
        .

      STOP
      END
```

SUBROUTINE: **RECV**

FORMAT: CALL RECV (channel, msg, num, flag)

channel Integer variable or constant specifying the identification number of a logical link to another processor.

msg Variable to store the received information.

num Integer variable or constant specifying the number of bytes to be received. It must be the same number of bytes which was sent.

flag Integer variable specifying the status of the receive operation.

DESCRIPTION:

RECV synchronously accesses information which was previously sent or broadcast by another processor. The **channel** declared by the sender of the information must be the same as the channel used by the receiver of the information. **Msg** can be any variable in the FORTRAN program. It can be an array name, or even a section of an array. **Num** indicates the number of bytes to be received, which must be the same number of bytes that was sent. Num must be less than 64 kbytes. The number of bytes for various FORTRAN data types is listed in Table 1 (see page 63).

When the RECV subroutine is executed for a particular channel, that channel is polled to determine if a message exists. If a message is available, and the number of bytes of available information is num, then the information is transferred to variable msg, and **flag** is set to one. If a message is available but num bytes are not available, then the entire message has not reached the receiving processor. Flag is set to zero, indicating that the information is unavailable to the programmer. Flag is also set to zero when a message does not exist. Information is written to msg only when num bytes of information are available at the time RECV is executed. The RECV subroutine is illustrated in Example 1 on page 27.

On occasion, information may be expected across several channels. Rather than processing each channel in sequence, it may be desirable to process information in the order in which it becomes available. RECV is capable of probing a channel to determine if information is available. If the information is available, it is received from that channel and processed. If the information on that channel is not available, then the other channels are scanned. It is not necessary to wait for information on one channel while information sits idle on another channel. This scenario is illustrated in Example 2 on page 28.

EXAMPLE 1 for RECV

N0P3

```
      PROGRAM MAIN1
      DIMENSION X(10)
      INTEGER*2 IVAL

        .
        .
      CALL OPENCH (2,3,2)
      CALL SEND (2,X,40,IFADDR)
C     Query to determine whether the
C     transfer has been initiated


10    IVAL = WORD (IFADDR)
      IF (IVAL .EQ. 0)
      THEN WRITE (1,100)
100   FORMAT (" Message
      initiated")
      ELSE GOTO 10

C     Variable X can now be
C     altered if so desired
      X(1)= 100.23

      STOP
      END
```

N3P2

```
      PROGRAM MAIN2
      DIMENSION Y(10)

      CALL OPENCH (2,0,3)

C     Query to determine if
C     message has been received

10    CALL RECV (2,Y,40,IFLG)
      IF (IFLG .NE. 0) GOTO 20
      WRITE (1,100)
100   FORMAT (1X,"Message not
      received")
      GOTO 30


C     Variable Y can now be accessed

20    Z= 2*Y(3)-C

        .
        .

30    STOP
      END
```

EXAMPLE 2 for RECV

```
      PROGRAM MAIN1
      DIMENSION X(100), Y(8), Z(39)
      INTEGER STAT1, STAT2, STAT3

          .
          .
          .

      ICNT= 0

10    CALL RECV (1,X,400,STAT1)
      IF (STAT1 .EQ. 1)
         CALL PROC_C1 (X)
         ICNT= ICNT + 1
         IF (ICNT .EQ. 3) GOTO 20
      ENDIF

      CALL RECV (2,Y,32,STAT2)
      IF (STAT2 .EQ. 1)
         CALL PROC_C2 (Y)
         ICNT= ICNT + 1
         IF (ICNT .EQ. 3) GOTO 20
      ENDIF

      CALL RECV (3,Z,156,STAT3)
      IF (STAT3 .EQ. 1)
         CALL PROC_C3 (Z)
         ICNT= ICNT + 1
         IF (ICNT .EQ. 3) GOTO 20
      ENDIF

      GOTO 10

20    CONTINUE
          .
          .
          .
      STOP
      END
```

SUBROUTINE: **RECVW**

FORMAT: CALL RECVW (channel, msg, num)

channel Integer variable or constant specifying the identification number of a logical link to another processor.

msg Variable to store the received information.

num Integer variable or constant specifying the number of bytes to be received. It must be the same number of bytes which was sent.

DESCRIPTION:

RECVW synchronously accesses information which was previously sent or broadcast by another processor. The **channel** declared by the sender of the information must be the same as the channel used by the receiver of the information. **Msg** can be any variable in the FORTRAN program. It can be an array name, or even a section of an array. **Num** indicates the number of bytes to be received, which must be the same number of bytes that was sent. Num must be less than 64 kbytes. The number of bytes for various FORTRAN data types is listed in Table 1 (see page 63).

When the RECVW subroutine is executed for a particular channel, that channel is polled to determine if a message exists. If a message does not exist, the processor continues to poll, waiting for a message to become available. If a message is available, and the number of bytes of available information is num, then the information is transferred to variable msg. If a message is available but num bytes are not available, then the entire message has not reached the receiving processor. The processor continues to wait until the remaining information becomes available. Once num bytes are available, the information is transferred to variable msg. Information is written to msg only after num bytes of the information are available.

A timeout mechanism is built into RECVW to avoid a deadlock in the event that the information never becomes available. If a timeout should occur, an advisory message is sent to the operating system, and program execution terminates.

An example follows.

EXAMPLE for RECVW

N0P3

```
      PROGRAM MAIN1
      DIMENSION X(10)

          .
          .
      CALL OPENCH (2,3,2)
      CALL SENDB (2,X,40)

C     Variable X can now be
C     altered if so desired
      X(1)= 100.23

      STOP
      END
```

N3P2

```
      PROGRAM MAIN2
      DIMENSION Y(10)

      CALL OPENCH (2,0,3)

      CALL RECVW (2,Y,40)

C     Variable Y is now
C     available for use
      Z= 2*Y(3)-C

          .
          .

      STOP
      END
```

SUBROUTINE: **BRDCST**

FORMAT: CALL BRDCST (channel, msg, num, bcode)

channel Integer variable or constant specifying the identification number of a logical link among several processors.

msg Variable to be transferred.

num Integer variable or constant specifying the number of bytes to transfer.

bcode Integer variable or constant specifying a broadcast code. Currently bcode = 2 is the only valid broadcast code.

DESCRIPTION:

BRDCST transmits a single message to many processors as efficiently as possible, using a broadcast algorithm which is built into the MPK. This algorithm transmits a message from one processor to all processors specified by a broadcast code, **bcode**. Several broadcast codes are provided for in the MPK; however, the Parallel Processing Library uses bcode = 2, which specifies that a message is broadcast to all computational processors.

A **channel** declared by the broadcaster is considered a global logical link to all other processors. Because of the global nature of this channel, non-valid entries of -1 are supplied as OPENCH parameters node id and proc id. These entries indicate that this processor is broadcasting a message to multiple processors on the same broadcast channel. All processors involved in a specific broadcast must use the same channel id number. This channel number cannot be used by processors which are not involved in a broadcast. In particular, a channel which is set up for the purpose of a broadcast cannot be used for an individual send/receive.

Msg can be any variable in the FORTRAN program. It can be an array name, or even a section of an array. **Num** indicates the number of bytes to transfer, and must be less than 64 kbytes. The number of bytes for various FORTRAN data types is listed in Table 1 (see page 63).

BRDCST is similar to a SENDB in that a message is composed containing a copy of the information to be transferred. The programmer does not have to wait until the transfer is initiated before altering the information.

As the broadcast message is received by each individual processor, it is treated by each processor as if it were individually sent. A receive operation may access the

information from the appropriate location as soon as it becomes available.

The situation may occur where a processor wants to transmit messages to a large number of processors, but not to all of them. At some point, it would be more efficient to broadcast that message, rather than sending it individually to so many processors. In such a case, the message would be sent to a few processors which never performed an OPENCH, RECVW, or RECV for that particular channel. Rather than generating an error message, the broadcast message is ignored by the few processors which should not receive it. Note that this situation could actually indicate an error, but this error will be overlooked for the sake of efficiency provided by the broadcast situation.

An example follows.

EXAMPLE for BRDCST

N0P3

```
      PROGRAM BRDPROC
      DIMENSION X(10)
        .
        .
C     Open broadcast channel
      CALL OPENCH (31,-1,-1)

C     Broadcast array X to all
C     computational processors
      CALL BRDCST (31,X,40,2)

      STOP
      END
```

N3P2

```
      PROGRAM RCVPROC1
      DIMENSION Y(10)

C     Open channel to N0P3
      CALL OPENCH (31,0,3)

      CALL RECVW (31,Y,40)

C     Variable Y can now be accessed

      Z= 2*Y(3)-C

      STOP
      END
```

N2P2

```
      PROGRAM RCVPROC2
      DIMENSION Y(10)

C     Open channel to N0P3
      CALL OPENCH (31,0,3)

      CALL RECVW (31,Y,40)

C     Variable Y can now be accessed
      Z= 2*Y(3)-C

      STOP
      END
```

N1P2

```
      PROGRAM RCVPROC3
      DIMENSION Y(10)

C     Open channel to N0P3
      CALL OPENCH (31,0,3)

      CALL RECVW (31,Y,40)

C     Variable Y can now be accessed
      Z= 2*Y(3)-C

      STOP
      END
```

SUBROUTINE: **READM**

FORMAT: CALL READM (node id, proc id, svar, dvar, num)

node id	Integer variable or constant specifying the source node.
proc id	Integer variable or constant specifying the source processor.
svar	Variable representing the information source.
dvar	Variable representing the information destination.
num	Integer variable or constant specifying the number of bytes to read.

DESCRIPTION:

READM asynchronously accesses information from another processor. This information access does not require a response from the processor whose data is being read; only the reading processor is actively involved in the read operation. Parameters **node id** and **proc id** specify the processor from whom data is to be read, while **num** indicates the number of bytes to transfer. Num must be less than 64 kbytes. The number of bytes for various FORTRAN data types is listed in Table 1 (see page 63).

A direct information transfer requires that the physical location of the information source and destination must be known. The programmer uses mapped FORTRAN common blocks in order to relate variables on one processor to variables on another. This is described on page 19. **Svar** represents the variable which is to be read. Although it is given in terms of the variable name on the reading processor, it may have a different name on the processor where the data is actually located. **Dvar** represents the variable which is to receive the data which has been read.

It is the responsibility of the FORTRAN programmer to guarantee that events are synchronized appropriately. Because of the asynchronous nature of this subroutine, it is typical to attach an extra element to the data being read, which is used as a flag to indicate whether the data has been received. A simple example is illustrated below. In this case, the array Z (or E in terms of N0P3) contains 100 data elements, and the 101st element is used as a flag to determine whether the data has been read.

It must be noted that if READM is used to read a variable which exists in another processor's shared memory area, then that shared data area must be

located on the processor from whom that data is being read. If the value being read is to be stored in a shared area by the reader, then that shared data area must be located on the processor performing READM. If the shared variable is located elsewhere, READM does not perform correctly. A synchronous send/receive combination may be used in this situation to avoid transfer errors.

EXAMPLE for READM

N1P2

```
      PROGRAM MAIN1
      COMMON /A/ Z(101)

         .
         .
         .

C     Clear read flag
      Z(101)= 0

      CALL READM (0,3,Z,Z,404)

C     Wait for data to become available
      ICNT= 100
20    IF (Z(101).NE.0) GOTO 30
      ICNT= ICNT-1
      IF (ICNT .GT. 0) GOTO 20
      GOTO 40

C     Variable Z can now be accessed
30    X= 2*Z(I)-D

40    STOP
      END
```

N0P3

```
      PROGRAM MAIN2
      COMMON /B/ E(101)

      DO 10 I=1, 100
10    E(I)= 3*(F+G)

C     Set read flag to indicate that
C     data is available
      E(101)= 1

         .
         .
         .
      STOP
      END
```

SUBROUTINE: **WRITEM**

FORMAT: CALL WRITEM (svar, node id, proc id, dvar, num, flag addr)

svar	Variable to be transferred.
node id	Integer variable or constant specifying the destination node.
proc id	Integer variable or constant specifying the destination processor.
dvar	Variable representing the information destination.
num	Integer variable or constant specifying the number of bytes to write.
flag addr	Integer variable which is returned to the user. It contains the address of a word which is cleared once the transfer is initiated.

DESCRIPTION:

WRITEM asynchronously transfers information to another processor. This information transfer does not require the assistance of the destination computational processor. **Svar** can be any variable in the FORTRAN program. It can be an array name, or even a section of an array. **Node id** and **proc id** specify the processor to which the data is to be written, while **num** indicates the number of bytes to transfer. Num must be less than 64 kbytes. The number of bytes for various FORTRAN data types is listed in Table 1 (see page 63).

A direct information transfer requires that the physical location of the information source and destination must be known. The programmer uses mapped FORTRAN common blocks to relate variables on one processor to variables on another. This is described on page 19. **Dvar** can be any variable in the FORTRAN program. It also can be an array name, or even a section of an array.

The programmer is responsible for guaranteeing that information exists before it is accessed, or that information has been transferred before it is altered. The writing processor can poll the word to which **flag addr** points to determine whether the transfer has been initiated. When this memory location is clear, the transfer has been initiated. This is similar to the synchronous SEND subroutine. The information transferred using WRITEM cannot be altered until it is sure that the transfer has been initiated (i.e., the word pointed to by flag addr is clear). Note that the term word implies a two-byte memory location. The example below includes a test sequence which is recommended in order to poll this variable correctly. The function WORD is supplied by FORTRAN 77 (ABSOFT, 1986).

When data is written to another processor, that processor is not notified that the data exists. It is the responsibility of the FORTRAN programmer to guarantee that events are synchronized appropriately. Because of the asynchronous nature of this subroutine, it is typical to attach an extra element to the data being written, which is used as a flag to indicate to the receiving processor that the information exists. A simple example is illustrated below. In this case, the array U (or F in terms of N0P3) contains 20 data elements, and the 21st element is used as a flag to determine whether the data has been written.

It must be noted that if WRITEM is used to write a variable which exists in a shared memory area, then that shared data area must be located in the processor performing WRITEM. If the value being written is to be stored in a shared area on the destination, then that shared data area must be located on the processor where the data is being written. If the shared variable is located elsewhere, WRITEM does not perform correctly. A synchronous send/receive combination may be used in this situation to avoid transfer errors.

EXAMPLE for WRITEM

N1P2

```
      PROGRAM MAIN1
      COMMON /A/ X,Y,Z(100)
      COMMON /C/ T,U(21),V
      INTEGER*2 IVAL

C     Set write flag
      U(21)= 1

C     Write variable U to N0P3
      CALL WRITEM (U,0,3,U,84,IFADDR)

C     Query to determine whether
C     the transfer has been initiated

 10   IVAL = WORD (IFADDR)
      IF (IVAL .EQ. 0)
      THEN WRITE (1,100) 30
100   FORMAT (" Message
     1          initiated")
      ELSE GOTO 10

C     Variable U can now be
C     altered if so desired
      U(2)= 164.3

      STOP
      END
```

N0P3

```
      PROGRAM MAIN2
      COMMON /B/ C,H,E(100)
      COMMON /D/ A,F(21),G

C     Clear write flag
      F(21)= 0
           .
           .
           .
      ICNT= 100
 10   IF (F(21) .EQ. 1) GOTO 20
      ICNT= ICNT-1
      IF (ICNT .GT. 0) GOTO 10
      GOTO 30

 20   T= F(4) * 2.3

 30   STOP
      END
```

MISCELLANEOUS SUBROUTINES

Timer Operations and Processor Identification

A few miscellaneous subroutines have been included in the Parallel Processing Library to assist the FORTRAN programmer.

Several subroutines involve accessing and reading a timer. Subroutines which perform timer operations are:

- TRSET
- TRSTRT
- TRSTOP
- TRREAD

Other miscellaneous subroutines included in the library assist in identifying particular processors. These subroutines are:

- NODE
- PROC
- GRAY
- GINV

Subroutines

A detailed description of each miscellaneous subroutine included in the Parallel Processing Library follows.

SUBROUTINE: **TRSET**

FORMAT: CALL TRSET (pid, ivar)

pid Integer variable or constant specifying a processor whose timer is to be set. This must be a communication processor.

ivar Integer array of timer characteristics. The elements of this array are described below.

ivar
(1) Timer type (=0 MIZAR 8115)
(2) Flag to change timer frequency
(3) Frequency range code
(4) Frequency value code
(5) Flag to change timer initial value
(6) Timer initial value

DESCRIPTION:

TRSET initializes the timer of communication processor **pid.** The timer characteristics to be initialized are specified by parameter **ivar.**

Currently only a timer on a communication processor (MIZAR 8115) can be manipulated, therefore ivar(1) must equal zero. All other values of ivar(1) are currently invalid. Should other timers be added for user access, array ivar will be altered to reflect the addition. The MIZAR timer and its parameters are described in detail in the *MIZAR 8115 CPU Module User's Manual* (Mizar, 1986).

Parameter ivar(2) indicates whether new frequency values are to be used for a specific timer. If ivar(2)= 1, then a new frequency range (ivar(3)) and a new frequency value (ivar(4)) must be supplied. These values can be taken from Table 2 on page 63. If ivar(2) = 0, then the default values of frequency range and frequency value are used. These default values are noted with an asterisk in Table 2 (see page 63).

Parameter ivar(5) indicates whether a new timer initial value is to be loaded. If ivar(5)= 1, then the new value is supplied in ivar(6). This parameter can have a value in the range $2 \leq$ ivar(6) $\leq$ 65535. If ivar(5) = 0, then a default value of 65535 is used.

A timer must be set before it is started or read. A processor cannot set a timer on another node. An example follows.

EXAMPLE for TRSET

N0P4

```
      PROGRAM TIMEPRG
      DIMENSION IVAR (6)

      IVAR(1)= 0
      IVAR(2)= 1

C Select baud range
      IVAR(3)= 16
C Set for 300 baud
      IVAR(4)= 68

      IVAR(5)= 1
C Initial counter value
      IVAR(6)= 4096
      CALL TRSET (2, IVAR)
      .
      .
      STOP
      END
```

SUBROUTINE: **TRSTRT**

FORMAT: CALL TRSTRT (pid)

pid Integer variable or constant specifying the processor whose timer is to be started. It must be a communication processor.

DESCRIPTION:

TRSTRT starts the timer of the local processor specified by **pid.** The timer will continue to count down until it is stopped, or until a timeout interrupt is generated. If a timeout should occur, an advisory message is sent to the operating system, and program execution terminates.

For an example, see page 45.

SUBROUTINE: **TRSTOP**

FORMAT: CALL TRSTOP (pid)

pid Integer variable or constant specifying the processor whose timer is to be stopped. It must be a communication processor.

DESCRIPTION:

TRSTOP stops the timer of the local processor specified by **pid.** Every timer which is started must be stopped in order to prevent a timeout interrupt from occurring. If a timeout should occur, an advisory message is sent to the operating system, and program execution terminates.

For an example, see page 45.

SUBROUTINE: **TRREAD**

FORMAT: CALL TRREAD (pid, tvar)

pid Integer variable or constant specifying the processor whose timer is to be read. It must be a communication processor.

tvar Integer variable which returns the timer value.

DESCRIPTION:

TRREAD reads the timer of the local processor specified by **pid.** This timer value is returned to the FORTRAN program in variable **tvar.** TRREAD is illustrated in the example below, along with other timer subroutines. Note that the timer should not be stopped until after all TRREADs have been performed.

In the example below, the timer is set at a frequency of 300 BAUD using TRSET. Note that since this is a count down timer, the second timer value is subtracted from the first in order to calculate the number of clock ticks which have transpired. The number of seconds is computed as the number of clock ticks multiplied by one over the BAUD rate. This example illustrates how to calculate timer overhead, and how to subtract it from the timing calculations.

EXAMPLE for TRSET, TRSTRT, TRSTOP, and TRREAD

N0P4

```
      PROGRAM TIMEPRG
      DIMENSION IVAR (6)

      IVAR(1)= 0
      IVAR(2)= 1
      IVAR(3)= 16
      IVAR(4)= 68
      IVAR(5)= 1
      IVAR(6)= 4096
      CALL TRSET (1, IVAR)

      BAUD= 300.0
      X= 1.0 / BAUD

      CALL TRSTRT (1)
      CALL TRREAD (1,ITIME1)
      CALL TRREAD (1,ITIME2)
    C Calculate overhead of the timer calls
      IOVER= ITIME1-ITIME2

      CALL TRREAD (1,ITIME1)
      (calculation to be timed)
      CALL TRREAD (1,ITIME2)
      CALL TRSTOP (1)

    C Calculate clock ticks
      ICALC=ITIME1-ITIME2-IOVER
    C Calculate seconds
      FTIME= FLOAT(ICALC)*X

      WRITE (1,10) ICALC, FTIME
10    FORMAT (1X,"Calculation required",1X,I5,1X, "clock ticks, which
      is",1X,F10.5,1X, "seconds.")

      STOP
      END
```

SUBROUTINE: **NODE**

FORMAT: CALL NODE (ndid)

ndid Integer variable which returns the current node id.

DESCRIPTION:

NODE accesses the node id of the current processor. This value is returned to the FORTRAN program in variable **ndid.**

For an example, see page 48.

SUBROUTINE: **PROC**

FORMAT: CALL PROC (pid)

pid Integer variable which returns the current processor id.

DESCRIPTION:

PROC accesses the processor id of the current processor. This value is returned to the FORTRAN program in variable **pid.**

EXAMPLE for NODE and PROC

```
      PROGRAM IDENT
          .
          .

      CALL NODE (IND)
      CALL PROC (IPR)

      WRITE (1,10) IND, IPR
   10 FORMAT (1X,"Greetings from node",1X,I2,1X,"processor",1X,I2)

      STOP
      END
```

SUBROUTINE: **GRAY**

FORMAT: CALL GRAY (ipos, igray)

ipos Integer variable or constant specifying the number to be converted to its gray code equivalent.

igray Integer variable which returns the gray code for **ipos**.

DESCRIPTION:

GRAY calculates the **ipos**th integer in the binary reflected gray code. This value is returned to the FORTRAN program in variable **igray**. This subroutine, along with GINV, assists in identifying nodes during program execution. Both of these subroutines involve the binary reflected gray code, which orders node numbers so that consecutive gray code numbers are adjacent nodes in the hypercube interconnection scheme. These subroutines enable a user to map a ring structure within the Hypercluster.

Table 3 lists the binary reflected gray code (see page 64). Figure 2 illustrates a ring mapped within a 3-D cube (see page 66).

For an example, see page 51.

SUBROUTINE: **GINV**

FORMAT: CALL GINV (inode, iloc)

inode Integer variable or constant specifying the node id for which a corresponding gray code location is to be found.

iloc Integer variable which returns the location of **inode** in the gray code ordering.

DESCRIPTION:

GINV calculates the location of **inode** in the gray code ordering. This value is returned to the FORTRAN program in variable **iloc**. This subroutine, along with GRAY, assists in identifying nodes during program execution. Both of these subroutines involve the binary reflected gray code, which orders node numbers so that consecutive gray code numbers are adjacent nodes in the hypercube interconnection scheme. These subroutines enable a user to map a ring structure within the Hypercluster.

Table 3 lists the binary reflected gray code, along with the location of each id in that code (see page 64). Figure 2 illustrates a ring mapped within a 3-D cube (see page 66).

EXAMPLE for GRAY and GINV

```
      PROGRAM IDENT

      CALL NODE (IND)

C     Calculate the location of IND in the gray code ordering
      CALL GINV (IND, IGINV)

C     Calculate the gray code of the location following IGINV.
C     IRN is the right neighbor of IND in a ring.
      CALL GRAY (IGINV+1, IRN)

C     Calculate the gray code of the location preceding IGINV.
C     ILN is the left neighbor of IND in a ring.
      CALL GRAY (IGINV-1, ILN)

      WRITE (1,10) IND,ILN,IRN
   10 FORMAT (1X,"Node",1X,I2,1X,"Left",1X,I2,1X,"Right",1X,I2)

      STOP
      END
```

APPLICATION

A simple programming example is included in this manual to illustrate how to use subroutines from the Parallel Processing Library. This example also gives a new user an indication of how problems can be partitioned using the Hypercluster architecture.

The example is a simple heat flow problem (Gerald, 1980) which uses Laplace's equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

to determine the steady state temperature of a flat plate exposed to constant boundary conditions. The problem is stated as follows:

> A thin steel plate is a 20 x 20 cm square. If one edge is held at 100°C, and the others are held at 0°C, what are the steady state temperatures at interior points?

The flat plate has a certain initial temperature, and is subjected to a temperature at each of its four boundaries (see Figure 3, page 67). What follows is a description of how this problem is discretized and mapped onto the Hypercluster.

To discretize this problem, the flat plate is divided into sections of size Δx by Δy. In this application, it is assumed that $\Delta x = \Delta y$. The resulting grid is now numbered so that each intersection represents a point of the flat plate at which a temperature can be evaluated (see Figure 4, page 68). A 14 x 14 grid is used for this example.

A central difference approximation is used to calculate the temperature $T_{i,j}$ at each interior grid point. All exterior grid points assume the boundary temperatures. The iterative equation

$$T_{i,j}^{(t+1)} = \frac{T_{i+1,j}^{(t)} + T_{i-1,j}^{(t)} + T_{i,j+1}^{(t)} + T_{i,j-1}^{(t)}}{4}$$

is used to calculate a temperature solution for each grid point (i,j). The value of $T_{i,j}$ at iteration (t+1) is an average of the temperatures of its four nearest neighbors at iteration (t). A solution is obtained when no grid point temperature changes its value as a result of performing an iteration.

This problem can be solved using a traditional computer. As grid sizes increase, however, the amount of memory and the computational time required to solve this problem may become too large for the traditional computer. Since the temperature value at each grid point is calculated from only the neighboring grid point values, the problem can easily be divided among multiple processors. The division of the problem is illustrated in Figure 5 (see page 69). Note that at boundaries, columns are repeated to accomodate data transfers.

A program has been written for the Hypercluster to execute this problem using a variable number of processors. Each computational processor executes an identical program which is included below. Currently the program runs on one processor per node, but can easily be extended to multiple processors per node. One processor (node= IOUTN, processor= IOUTP) is singled out as a "controlling" processor, which performs I/O and convergence tests. This processor is identified in the FORTRAN PARAMETER statement.

A ring is mapped within the Hypercluster, and the computational grid is mapped to each processor as illustrated in Figure 5. The ring structure is determined within the program using library subroutines NODE, PROC, GINV, and GRAY. Each processor sends the boundary values of its portion of the grid to its neighbors; subroutines SENDB and RECVW are used to accomplish this data transmission. Each processor then performs its own calculation for a particular iteration, and determines its own convergence.

The individual convergence results are transmitted to the controlling processor on every tenth iteration to determine the convergence of the entire problem. Subroutines SENDB, RECVW, and BRDCST are used to transmit convergence information among processors. Once convergence is determined, each processor transmits its section of the grid temperature values to the controlling processor, which writes the information to an output file. The output file for this application appears on page 60. Note again that columns are repeated at the boundaries.

Implementing this application requires a significant amount of communication because of the convergence test. A more economical method could be used in place of the method demonstrated here. Subroutines to time the code performance have been included in this application. Timing results are illustrated in Table 4 (see page 64).

Application

```
      PROGRAM CALCTEMP

C PARAMETER DESCRIPTION:

C   IMAX    GRID DIMENSION IN THE Y DIRECTION
C   JMAX    GRID DIMENSION IN THE X DIRECTION
C   IPROC   PROCESSOR ID OF THE PROCESSOR WITHIN EACH NODE ON
C           WHICH TO EXECUTE THE CODE
C   NODES   NUMBER OF NODES BEING USED FOR COMPUTATION
C   IOUTN   NODE ID OF THE CONTROLLING PROCESSOR
C   IOUTP   PROCESSOR ID OF THE CONTROLLING PROCESSOR
C   MAXIT   MAXIMUM NUMBER OF ITERATIONS
C   IPERND  NUMBER OF PROCESSORS USED PER NODE FOR
C           COMPUTATION

      PARAMETER (IMAX=14, JMAX=5, IPROC=4, NODES=4)
      PARAMETER (IOUTN=0, IOUTP=4, MAXIT=1500, IPERND=1)

      DIMENSION X(IMAX,JMAX), XT(IMAX,JMAX), IVAR(6)
      LOGICAL*1 CONV, RCNV, ICHK, EVEN, ICNTRL

C DETERMINE MY NODE AND PROCESSOR ID

      CALL NODE (IND)
      CALL PROC (IPR)

C CALCULATE MY RIGHT AND LEFT NEIGHBOR IN A RING

      CALL GINV (IND, IGINV)
      EVEN= MOD(IGINV,2) .EQ. 0
      CALL GRAY (IGINV+1,IRN)
      IF (IGINV .NE. 0) CALL GRAY (IGINV-1,ILN)

C IF I'M ON A BOUNDARY, SET THAT NEIGHBOR TO AN INVALID ID

      IF (IND .EQ. 0) ILN= -1
      IF (IRN .GE. NODES) IRN= -1

C DETERMINE IF I AM THE CONTROLLING PROCESSOR

      ICNTRL= (IOUTN .EQ. IND) .AND. (IOUTP .EQ. IPR)
      IF (ICNTRL) THEN
        OPEN (UNIT=2,ACTION='WRITE',ACCESS='SEQUENTIAL',
     *        FORM='FORMATTED')
```

```
C INITIALIZE TIMER

        IVAR(1)= 0
        IVAR(2)= 1
        IVAR(3)= 16
        IVAR(4)= 17
        IVAR(5)= 1
        IVAR(6)= 65535
        CALL TRSET (1,IVAR)
        BAUD= 110.
        F= 1./BAUD
        CALL TRSTRT (1)
        CALL TRREAD (1,ITIME1)
        CALL TRREAD (1,ITIME2)
        IOVER= ITIME1-ITIME2
      ENDIF

C DETERMINE APPROPRIATE CHANNELS TO REACH EACH NEIGHBOR

      IF (EVEN) THEN
        IRCH= 1
        ILCH= 2
      ELSE
        IRCH= 2
        ILCH= 1
      ENDIF

C OPEN CHANNELS TO VALID NEIGHBORS

      IF (IRN .NE. -1) CALL OPENCH (IRCH,IRN,IPROC)
      IF (ILN .NE. -1) CALL OPENCH (ILCH,ILN,IPROC)

C OPEN CHANNELS TO CONTROLLING PROCESSOR

      IF (ICNTRL) THEN
        DO 5 I= 0,NODES-1
        DO 5 J= IPROC, IPROC+IPERND-1
          IF ((IND .NE. I) .OR. (IPR .NE. J)) THEN
            ICH= I*IPERND+J
            CALL OPENCH (ICH, I, J)
          ENDIF
5       CONTINUE
        CALL OPENCH (31,-1,-1)
```

```
      ELSE
        IPCH= IND*IPERND+IPR
        CALL OPENCH (IPCH, IOUTN, IOUTP)
        CALL OPENCH (31,IOUTN,IOUTP)
      ENDIF

C INITIALIZE GRID TEMPERATURE VALUES  (BOUNDARIES)

      DO 10 J=1,JMAX
        X(1,J)= 0.
10      X(IMAX,J)= 0.
      IF (ILN .NE. -1) GOTO 20
      DO 15 I=1,IMAX
15      X(I,1)= 0.
20    IF (IRN .NE. -1) GOTO 30
      DO 25 I=1,IMAX
25      X(I,JMAX)= 100.

C INITIALIZE GRID TEMPERATURE VALUES  (INTERIOR POINTS)

30    DO 35 J=2,JMAX-1
      DO 35 I=2,IMAX-1
35      X(I,J)= 25.

C ADDITIONAL PARAMETERS:

C ICOL  NUMBER OF BYTES IN A COLUMN OF DATA TO BE
C       TRANSFERRED
C E     CONVERGENCE TOLERANCE
C IT    ITERATION COUNT

      ICOL= (IMAX-2)*4
      E= 0.001
      IT= 1

C ITERATION LOOP

      IF (ICNTRL) CALL TRREAD (1,ITIME1)

C  TRANSMIT DATA TO NEIGHBORS

40    IF (IRN .NE. -1) CALL SENDB (IRCH,X(2,JMAX-1),ICOL)
      IF (ILN .NE. -1) CALL SENDB (ILCH,X(2,2),ICOL)
```

```
C RECEIVE DATA FROM NEIGHBORS

      IF (IRN .NE. -1) CALL RECVW (IRCH,X(2,JMAX),ICOL)
      IF (ILN .NE. -1) CALL RECVW (ILCH,X(2,1),ICOL)

C PERFORM CALCULATION ON MY SECTION OF GRID, AND DETERMINE
C MY OWN CONVERGENCE

      CONV= .TRUE.
      DO 50 J=2,JMAX-1
      DO 50 I=2,IMAX-1
        XT(I,J)= (X(I+1,J)+X(I-1,J)+X(I,J+1)+X(I,J-1))/4.0
50      CONV= CONV .AND. (ABS(XT(I,J)-X(I,J)) .LE. E)

C UPDATE COMPUTATIONAL GRID WITH NEWLY CALCULATED VALUES

      DO 55 J=2,JMAX-1
      DO 55 I=2,IMAX-1
55      X(I,J)= XT(I,J)

C CHECK CONVERGENCE ON EVERY 10TH ITERATION

      ICHK= MOD(IT,10) .EQ. 0
      IF (ICHK) THEN

        IF (ICNTRL) THEN

C CALCULATE CONVERGENCE FOR ENTIRE PROBLEM

          DO 6 I= 0,NODES-1
          DO 6 J= IPROC, IPROC+IPERND-1
            IF ((IND .NE. I) .OR. (IPR .NE. J)) THEN
              ICH= I*IPERND+J
              CALL RECVW (ICH, RCNV, 1)
              CONV= CONV .AND. RCNV
            ENDIF
6         CONTINUE

C BROADCAST CONVERGENCE RESULT TO ALL COMPUTATIONAL
C PROCESSORS

          CALL BRDCST (31,CONV,1,2)

        ELSE
```

```
C SEND MY OWN CONVERGENCE INFORMATION TO THE CONTROLLING
C PROCESSOR

        CALL SENDB (IPCH,CONV,1)

C RECEIVE CONVERGENCE RESULT FOR THE ENTIRE PROBLEM

        CALL RECVW (31,CONV,1)

      ENDIF

     ELSE

      CONV= .FALSE.

     ENDIF

65   IF (CONV) THEN

      IF (ICNTRL) THEN

        CALL TRREAD (1,ITIME2)
        CALL TRSTOP (1)
        ICALC= ITIME1-ITIME2-IOVER
        FTIME= FLOAT(ICALC) * F
        RATE= FTIME/ FLOAT(IT)

C OUTPUT RESULTS FROM CONTROLLING PROCESSOR

        WRITE (2,91) IT
91      FORMAT (" PROGRAM COMPLETED IN ",I6," ITERATIONS.")
        WRITE (2,92) FTIME
92      FORMAT (" TIME REQUIRED: ",F6.2," SECONDS")
        WRITE (2,93) RATE
93      FORMAT (" RATE:  ",F8.4," SEC/IT",//)
        WRITE (2,94) IND, IPR
94      FORMAT (" DATA FROM NODE ",I2," PROCESSOR ",I2,": ",/)
        DO 200 I=1,IMAX
          WRITE (2,95) (X(I,J), J=1,JMAX)
95        FORMAT (10(1X,F6.2))
200     CONTINUE
        WRITE (2,96)
96      FORMAT (1X,//)
```

```
C RECEIVE AND OUTPUT RESULTS FROM REMAINING PROCESSORS

          DO 7 I= 0,NODES-1
          DO 7 J= IPROC, IPROC+IPERND-1
            IF ((IND .NE. I) .OR. (IPR .NE. J)) THEN
              ICH= I*IPERND+J
              CALL RECVW (ICH, XT, IMAX*JMAX*4)
              WRITE (2,94) I,J
              DO 201 IP=1,IMAX
                WRITE (2,95) (XT(IP,JP), JP=1,JMAX)
201           CONTINUE
              WRITE (2,96)
            ENDIF
7         CONTINUE

        ELSE

C SEND RESULTS TO THE CONTROLLING PROCESSOR FOR OUTPUT

          CALL SENDB (IPCH,X,IMAX*JMAX*4)

        ENDIF

      ELSE

C CONTINUE CALCULATION WITH THE NEXT ITERATION

        IT= IT+1
        IF (IT .LE. MAXIT) GOTO 40
        IF (ICNTRL) CALL TRSTOP (1)

      ENDIF

C WHEN CALCULATION IS COMPLETE, CLOSE I/O UNIT

110   IF (ICNTRL) CLOSE (UNIT=2)

      STOP
      END
```

Application

Program Completed: 110 iterations
Time Required: .93 seconds
Rate: .0084 sec/it

DATA FROM NODE 0, PROCESSOR 4

00.00	0.00	0.00	0.00	0.00
00.00	0.65	1.35	2.12	3.03
00.00	1.27	2.61	4.11	5.87
00.00	1.80	3.71	5.84	8.32
00.00	2.23	4.59	7.22	10.27
00.00	2.53	5.20	8.17	11.61
00.00	2.68	5.52	8.66	12.29
00.00	2.68	5.52	8.66	12.29
00.00	2.53	5.20	8.17	11.61
00.00	2.23	4.59	7.22	10.27
00.00	1.80	3.71	5.84	8.32
00.00	1.27	2.61	4.11	5.87
00.00	0.65	1.35	2.12	3.03
00.00	0.00	0.00	0.00	0.00

DATA FROM NODE 1, PROCESSOR 4

00.00	00.00	00.00	00.00	00.00
02.12	03.03	04.14	05.55	07.38
04.11	05.87	08.00	10.66	14.10
05.84	08.32	11.31	15.02	19.70
07.22	10.27	13.92	18.39	23.95
08.17	11.61	15.70	20.67	26.77
08.66	12.29	16.60	21.82	28.18
08.66	12.29	16.60	21.82	28.18
08.17	11.61	15.70	20.67	26.77
07.22	10.27	13.92	18.39	23.95
05.84	08.32	11.31	15.02	19.70
04.11	05.87	08.00	10.66	14.10
02.12	03.03	04.14	05.55	07.38
00.00	00.00	00.00	00.00	00.00

DATA FROM NODE 2, PROCESSOR 4

00.00	00.00	00.00	00.00	100.00
13.46	19.07	28.95	49.35	100.00
24.91	33.88	47.39	68.43	100.00
33.63	44.16	58.30	77.00	100.00
39.73	50.83	64.64	81.27	100.00
43.55	54.79	68.15	83.45	100.00
45.37	56.63	69.72	84.39	100.00
45.37	56.63	69.72	84.39	100.00
43.55	54.79	68.15	83.45	100.00
39.73	50.83	64.64	81.27	100.00
33.63	44.16	58.30	77.00	100.00
24.91	33.88	47.39	68.43	100.00
13.46	19.07	28.95	49.35	100.00
00.00	00.00	00.00	00.00	100.00

DATA FROM NODE 3, PROCESSOR 4

00.00	00.00	00.00	00.00	00.00
05.55	07.38	09.87	13.46	19.07
10.66	14.10	18.65	24.91	33.88
15.02	19.70	25.73	33.63	44.16
18.39	23.95	30.93	39.73	50.83
20.67	26.77	34.30	43.55	54.79
21.82	28.18	35.95	45.37	56.63
21.82	28.18	35.95	45.37	56.63
20.67	26.77	34.30	43.55	54.79
18.39	23.95	30.93	39.73	50.83
15.02	19.70	25.73	33.63	44.16
10.66	14.10	18.65	24.91	33.88
05.55	07.38	09.87	13.46	19.07
00.00	00.00	00.00	00.00	00.00

TABLES AND FIGURES

The following Tables and Figures are available for reference.

Tables

Figures

Table 1
Size of Fortran Data Types

FORTRAN data type	Number of bytes
Integer	4
Integer*2	2
Real	4
Double Precision	8
Logical	1

Table 2
MIZAR Timer Parameters

Baud rate	ivar(3)	ivar(4)
50	16	0
75	144	0
*110	16	17
134.5	16	34
150	144	51
200	16	51
300	16	68
600	16	85
1050	16	119
1200	16	102
1800	144	170
2000	144	119
2400	16	136
4800	16	153
7200	16	170
9600	16	187
19.2 K	144	204
38.4 K	16	204

Table 3
Binary-Reflected Gray Code

Id	GRAY	GINV
0	0	0
1	1	1
2	3	3
3	2	2
4	6	7
5	7	6
6	5	4
7	4	5

Table 4
Application Timing Results

Number of Processors	Grid size	Time (seconds)	Speedup
1	14 x 14	3.04	1.00
2	14 x 14	1.65	1.84
4	14 x 14	0.93	3.27
1	34 x 34	104.37	1.00
2	34 x 34	53.29	1.96
4	34 x 34	27.44	3.80

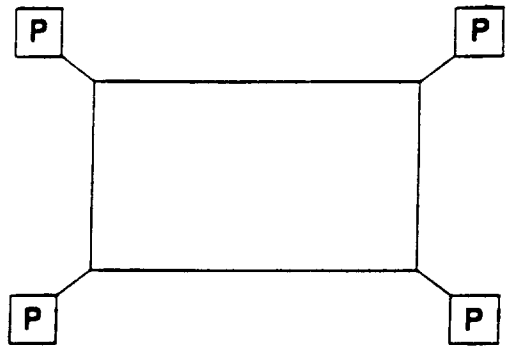


Figure 1a. 2-D hypercube

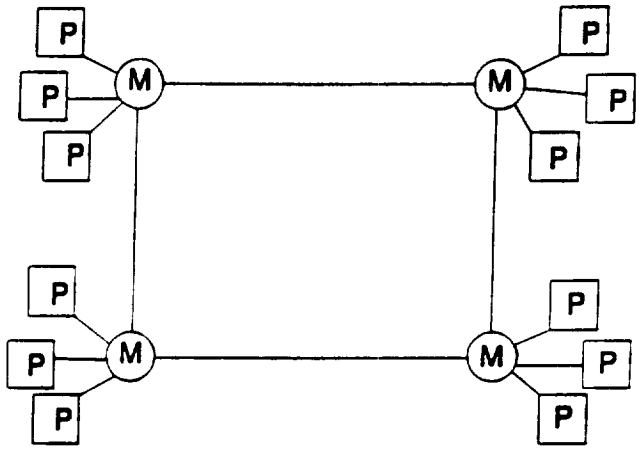


Figure 1b. 2-D Hypercluster

P= Processor
M= Shared Memory

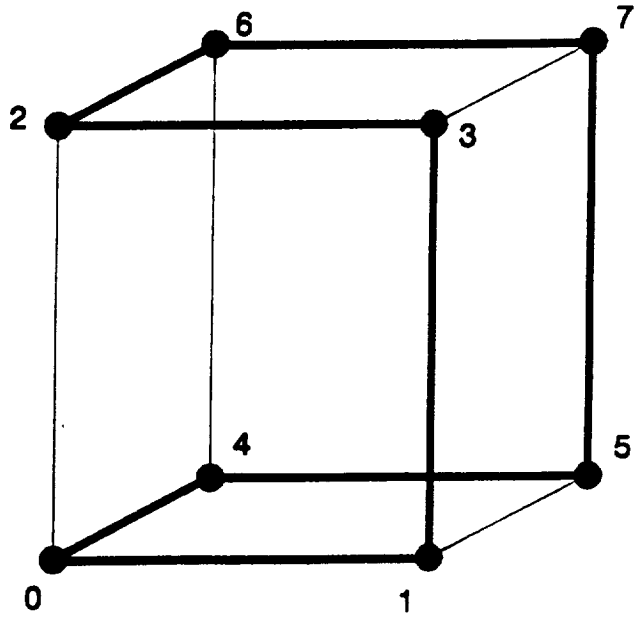


Figure 2. Ring mapped within a 3-D cube.

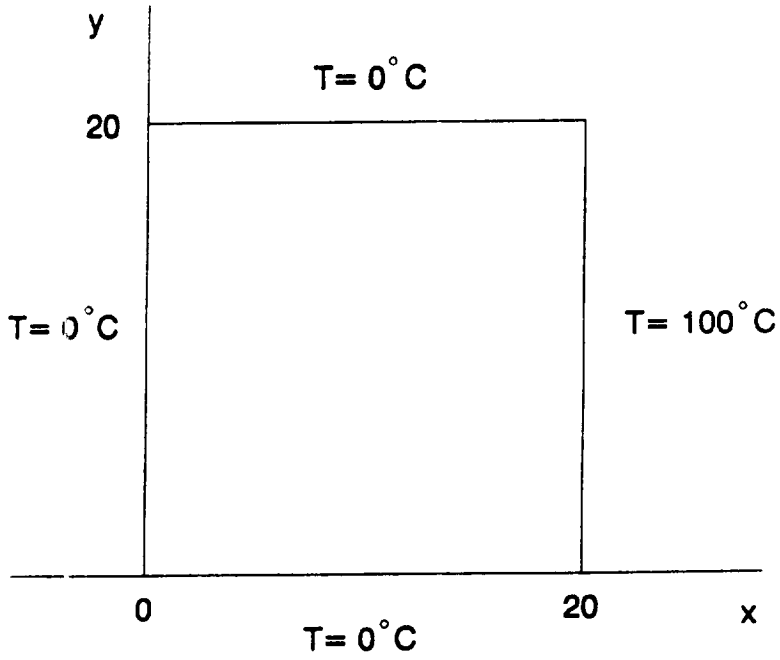
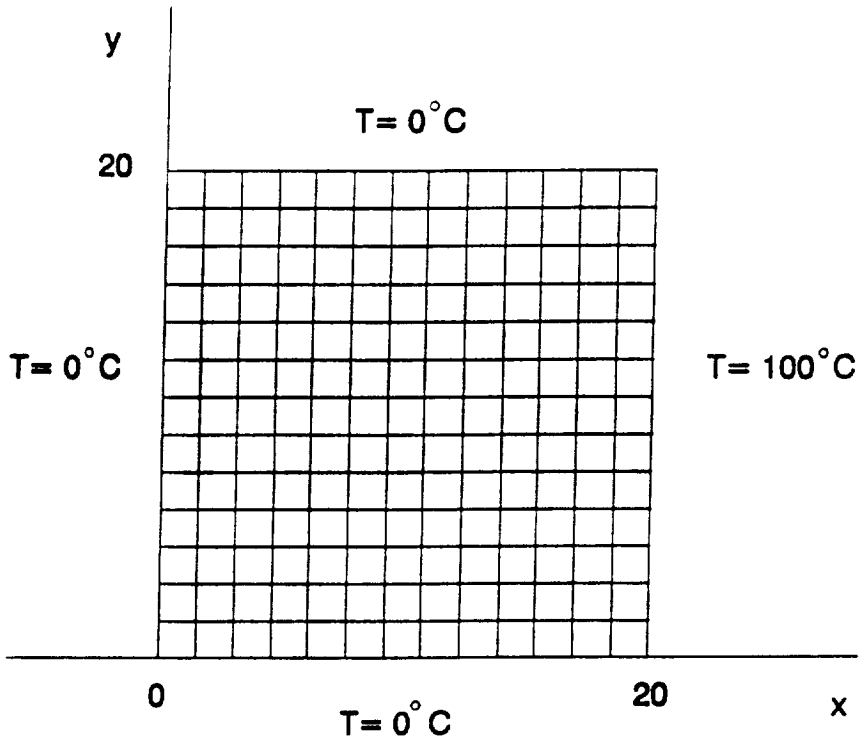


Figure 3. Thin steel plate which is 20 cm x 20 cm. One edge is held at 100° C. The other edges are held at 0°C.



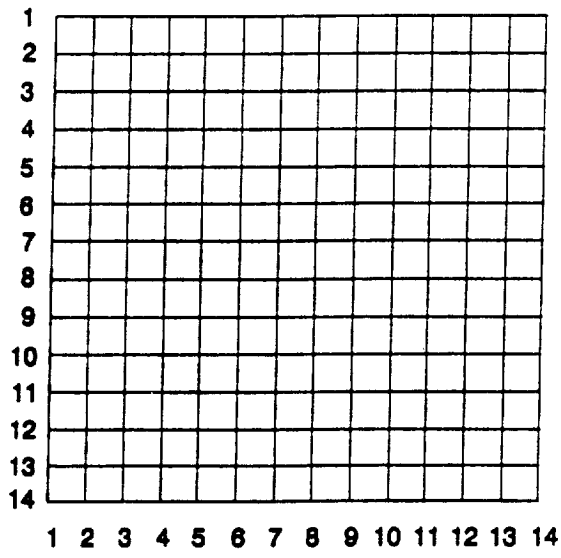
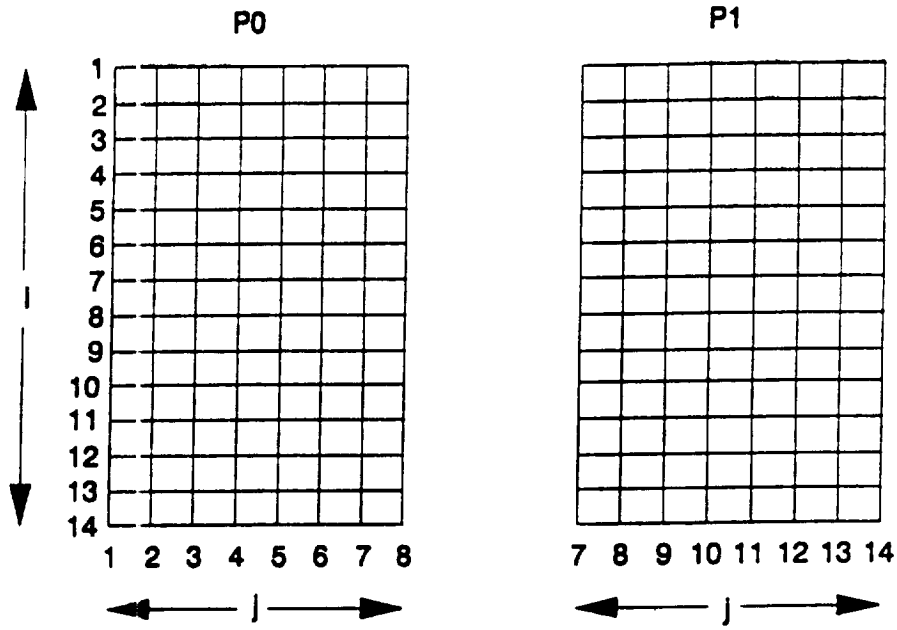
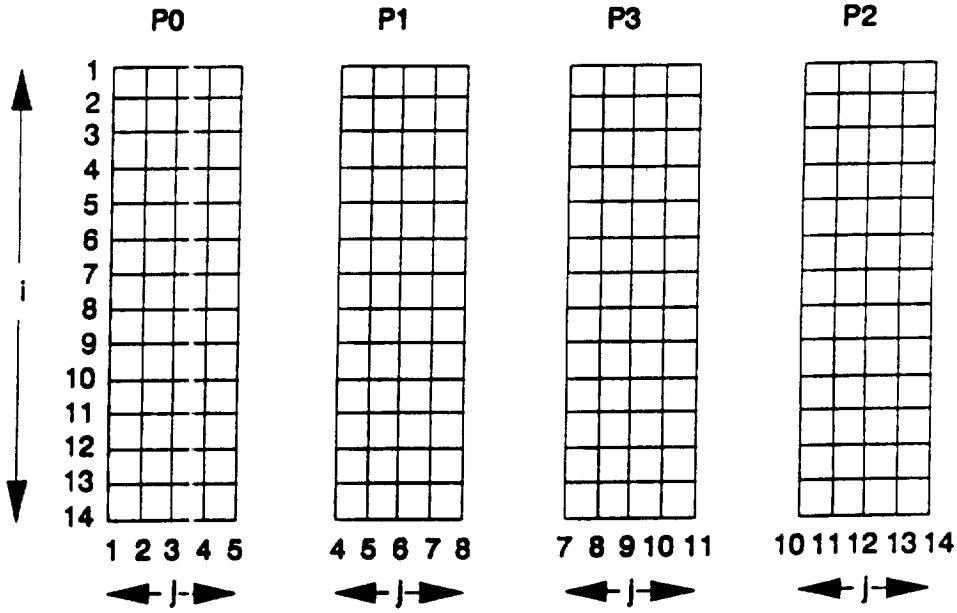


Figure 4: 14 x 14 computational grid



a. 2-node problem



b. 4-node problem

Figure 5. Division of problem for multiple nodes.

REFERENCES

The Hypercluster is a many-facetted architecture. A list of reference manuals is included here for users needing further details about the system.

- For more information on the MIZAR timers, refer to the *MZ 8115 CPU Module User's Manual,* Board Revision E, 1986.
- For information on the FORTRAN compiler, refer to the *FORTRAN 77 Compiler and Debugger Reference Manual,* V2.2 ABSOFT, Inc., 1986.
- For information on the vector processors, refer to *Warrior Reference Manual* SKY Computers, Inc., 1986 DOC#WAR-ALL-RM-86-1.2.
- For more information on the Hypercluster architecture and its rationale, refer to "The Hypercluster: A Parallel Processing Test-Bed Architecture for Computational Mechanics Applications" Blech, R. A., NASA Technical Memorandum 89823, July 1987.
- For the source of the heat flow application, refer to Gerald, Curtis F., *Applied Numerical Analysis*, 2nd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts, May 1980, pp. 340-356.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-185231		

4. Title and Subtitle	5. Report Date
Hypercluster Parallel Processing Library User's Manual	April 1990
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Angela Quealy	None
	10. Work Unit No.
	505-62-21

9. Performing Organization Name and Address	11. Contract or Grant No.
Sverdrup Technology, Inc. Lewis Research Center Group 2001 Aerospace Parkway Brook Park, Ohio 44142	NAS3-25266
	13. Type of Report and Period Covered Contractor Report Final

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Lewis Research Center Cleveland, Ohio 44135-3191	

15. Supplementary Notes

Project Manager, Richard A. Blech, Internal Fluid Mechanics Division, NASA Lewis Research Center.

16. Abstract

This User's Manual describes the Hypercluster Parallel Processing Library, composed of FORTRAN-callable subroutines which enable a FORTRAN programmer to manipulate and transfer information throughout the Hypercluster at NASA Lewis Research Center. Each subroutine and its parameters are described in detail. A simple heat flow application using Laplace's equation has been included to demonstrate the use of some of the library's subroutines. The manual can be used initially as an introduction to the parallel features provided by the library. Thereafter it can be used as a reference when programming an application.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel processing Parallel programming Software	Unclassified – Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price*
Unclassified	Unclassified	75	A04

*For sale by the National Technical Information Service, Springfield, Virginia 22161